HW/SW Codesign of an Engine Management System[*]

M. Baleani[†‡] A. Ferrari[‡] A. Sangiovanni-Vincentelli[‡§] C. Turchetti[†]

[†]Department of Electronics
University of Ancona
60131 Ancona, Italy
{*mbaleani,turchetti*}*@ee.unian.it*

[‡]PARADES EEIG
Via San Pantaleo, 66
00186 Rome, Italy
aferrari@parades.rm.cnr.it

[§]Department of EECS
University of California
Berkeley, CA 94709
alberto@eecs.berkeley.edu

Abstract

The design process for an engine management system is presented. The functional specification of the system has been captured using C and C++ as specification languages. The validation of the specification has been carried out using functional simulation. Then an architecture for the implementation of the functional specification is selected among a set of three possible alternatives, all based on the same micro-controller, characterized by different hardware-software trade-offs. The choice is motivated by a fast performance estimation that can also be used to identify the parts of the design that could be moved across the hardware-software partition to obtain better cost or better performance. The case study has been performed in the Felix VCC framework.

1. Introduction

Embedded systems are likely to become pervasive in all aspects of everyday life. Given the strong pressure on time-to-market and production cost, there is a growing interest in design methodologies that can substantially reduce design time and yet improve its quality, cost and reliability. There has been a flurry of activities in this domain that have yielded a number of environments, description languages, and algorithms for verification and synthesis of embedded systems. However, there is still a lack of experimental results that prove the effectiveness of the proposed approach. Our goal is to demonstrate how a fairly complex, real, application of embedded system design can be tackled within the framework described in [3]. The automotive domain represents an important share of the embedded systems' market, where complexity, hard real-time constraints, product customization and safety are ever increasing requirements. In this paper the case study of the full architectural/behavioral design of an engine management unit is presented. We focus on engine position acquisition and treatment, for which six different hw/sw architectures are devised and analyzed.

[*]This work has been partially funded by CNR – MADESS II project, code 2.1.3.

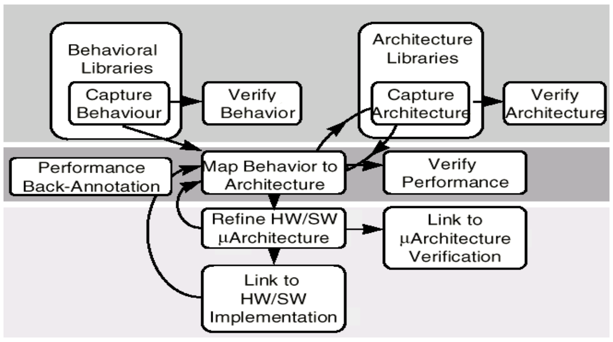
Behavioral Libraries
Capture Behaviour
Verify Behavior
Architecture Libraries
Capture Architecture
Verify Architecture
Performance Back-Annotation
Map Behavior to Architecture
Verify Performance
Refine HW/SW µArchitecture
Link to µArchitecture Verification
Link to HW/SW Implementation

Figure 1. The VCC design methodology

The basic tenets of the design methodology [1, 3] are presented pictorially in Figure 1.

Behaviors and Architectures. The system behavior is entered and analyzed. Parts of the overall behavior may be taken from an existing library of behavioral components. The analysis of system behavior allows to capture design errors early in the design phase. Analysis may be performed by simulation or by formal verification techniques.

In parallel, a class of architectures, i.e. a class of physical or ideal components, is selected. For example, a particular micro-processor architecture (e.g. 32-bit or 16-bit, RISC or CISC), a DSP, an interconnection scheme (e.g. the AMBA Bus), are chosen as candidates for the implementation of the behavior. These components may come from an existing library of architectural IPs or may be ideal models of components that will be designed later.

Architecture selection may be performed with different criteria in mind: from cost to reliability, from performance to commercial viability, thus allowing a careful examination of tradeoffs that are not purely technical but that can also incorporate business objectives.

Mapping. A critical step is mapping a behavior onto a candidate architecture, i.e. assigning behavioral functions and communication arcs to the architectural resources. At this point, partitioning between hardware and software takes place. When several behavioral blocks are mapped onto a programmable IP block, such as a micro-processor or a DSP, the corresponding functions are implemented as software tasks running on the processor. Since the tasks may be concurrent, the choice of a scheduling policy for the processor is needed. When a behavioral block is assigned to a dedicate hardware unit, that behavior is implemented directly in hardware. The mapping process establishes a set of relationships between the application behavior and the architecture on which it will be realized. The resulting system is then evaluated via performance analysis of speed, power and cost. In the VCC environment this is done by running behavioral simulation annotated with information derived from architectural estimated models representing the performance of the selected architecture. With early accurate performance analysis, a design where constraints are not met can be corrected early, re-designing the behavior to be implemented or changing the target architecture. Behavior specification, architecture selection and mapping are iterated and analyzed until a satisfactory solution is found.

Refinement. Upon completion of the mapping step, the architecture with the implemented behavior is successively refined to micro-architectural levels where detailed instruction sets, RTL models and programming languages are selected and tested for compliance with the higher level requirements. Since now the ideal components are better characterized, it may be necessary to back-annotate the higher level models with more accurate parameters, so that performance evaluation can be re-done with more confidence about the final outcome. The refined target architecture can finally be exported to external environments such as a ISS/logic simulator co-verification environment or a rapid-prototyping board.

2. Engine Management Systems

The overall goal of engine management systems is to offer appropriate driving performance (e.g. speed, comfort, safety) while minimizing fuel consumption and emissions. Engine management has at its disposal as engine control inputs: throttle position[1], fuel injection and spark ignition. While the throttle position is controlled at fixed sampling time, fuel injection and spark ignition require synchronization with the engine crankshaft position.

In this paper we address the specific problem of injection and ignition synchronization in 4-cylinder multi-point spark-ignition engines. In more detail, the term *synchronization* means the computation of the current position of each piston and the working phase (i.e. intake, compression, expansion or exhaust) of the related cylinder. Synchronization is essential for timing the opening of fuel injectors and the ignition of the spark plugs. For example, to burn completely the air/fuel mixture, the ratio between air and fuel should be close to the *stoichiometric ratio* and the spark should be fired with a certain advance with respect to the compression Top Dead Center (TDC). The supplied torque and the emitted pollutants depend crucially on the accuracy of these operations. Moreover, the measurement of the engine-position changing rate allows to compute the engine-revolution speed (RPM), which is a basic parameter for any engine control law.

The engine position (i.e. both the position of each piston and the working phase of the related cylinder) is represented by the *engine angle*, an angular reference with a 720° period corresponding to a complete engine cycle. The engine angle is computed by acquiring and processing two synchronization signals. The *fly-wheel signal* is provided by a Hall-effect sensor faced to a toothed wheel integral with the crankshaft. The wheel has 60 teeth, hence each tooth corresponds to a 6 gradient degree angle slice. Two teeth are missing, in order to create a reference needed for their numbering. The crankshaft angular reference (or *crankshaft angle*) has a 360° period: two crankshaft revolutions are performed for every engine cycle. The *phase signal* is provided by a Hall-effect sensor placed on the camshaft. It is faced to a wheel integral with the camshaft, whose revolution period corresponds to an engine cycle. The particular shape of the camshaft wheel yields two different logic levels for every successive crankshaft revolution, thus allowing to compute the engine angle value (i.e. the 720° angular reference) from the crankshaft angle one (see Figure 2).

Numbering fly-wheel teeth and computing the engine angle involves sensing the crankshaft signal edges, detecting the asymmetry and "building" the two missing teeth. Since synchronization is critical for the overall engine control strategy, the functional description contains an algorithm that guarantees robustness. A 6° error is allowed, provided that it is corrected within one crankshaft revolution. The system is instructed to filter the fly-wheel signal, to monitor engine stalls, to detect sensor failures (e.g. the loss of a tooth, the presence of glitches, the absence of transitions on either signal) and, in this case, to recover (if possible) system functionality, to re-synchronize the system or, in the worst case, to shut-down the engine.

3. Functional Description and Validation

The system model has been described using the *globally asynchronous locally synchronous* (GALS) formal

[1]In modern drive-by-wire systems.

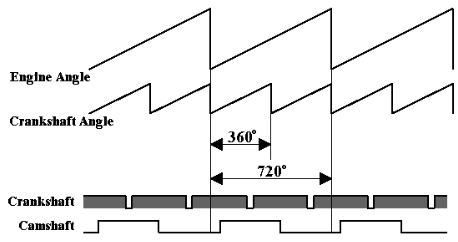

Figure 2. Crankshaft angle and engine angle

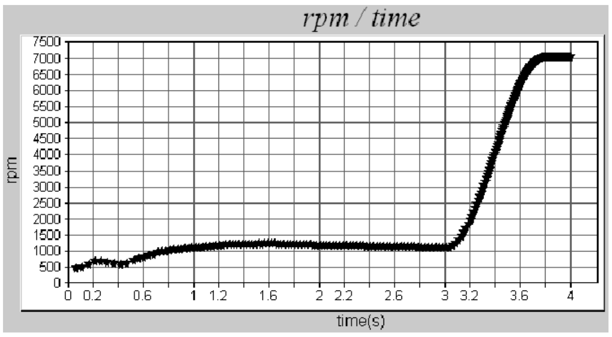

Figure 3. Simulated RPM profile

model [1], supported by the Felix/VCC environment.

The original behavioral block specifications, in essence extended FSMs with timeouts and period measurements, have been decomposed into pure functional blocks and pure time-related blocks (e.g. timers and free running counters). The functional blocks have the semantics of Co-design FSM [1] and are captured using a C subset, called PoindexterC. The timing blocks are captured using the full C++ language, which has been used also to describe test-bench blocks.

The overall functional specification, including also fault diagnosis and recovery, consists of 13 CFSMs, 3 down-counters and 2 free-running counters[2], for a total of about 2000 C/C++ code lines.

The behavioral description of the system has been validated performing extensive simulations. The functional requirements state that the error on the RPM calculation must be less than 0.2%, while the accuracy of the TDC event detection must be at least 0.1 gradient degree. These properties are verified for all the performed simulations using a C++ behavioral block, which is part of the system test-bench. Figure 3 shows the RPM profile as it is reconstructed by the system. Simulations are run not only under nominal conditions: we have accounted for every fault which may occur on any of the two sensors, as specified by the functional requirements.

Simulations use the real acquisition (sampled and quantized) of the output of the sensors of 4 seconds of engine management. As shown in Figure 3, the first 3 seconds refer to the engine's crank and idle phases, starting from engine start-up. This represents the most critical condition for synchronization since the synchronism has to be established and because of the maximum RPM changing rate. The idle phase is then followed by an acceleration (0.8s), bringing the engine up to 7000rpm, and a hold period of 0.2s at the maximum rpm value. In this way, we can assess the behavior of the system also under a maximum load condition, which is extremely important especially when coping with its implementation onto finite resources.

The system's simulation takes about 10 seconds on a Windows NT 4.0 PC with a 333MHz PentiumII CPU and a 96MB memory.

4. Architecture Definition

The target architecture is based on the Hitachi SH7055 micro-controller architecture [2]. Only the basic topology has been captured: not every communication path is described. Each architectural element has been defined as an abstract performance model with no implementation detail.

The hardware partition relies on two components: the CPU and the Advanced Time Unit (ATU). The CPU is a 32-bit load-store RISC architecture with a 40MHz maximum operating frequency and processes most instructions in a single clock cycle through a five-stage pipeline. In the VCC environment, the CPU model is characterized by a table of coefficients giving cycle counts for a basic set of generic or atomic operators. These common operators, such as *load to/from memory,store to memory*, *ALU operation*, *register move*, *branch/return*, map into actual processor instructions. This performance model does not take into account any micro-architectural detail. In order to account for the effects of pipeline stalls, we may use average performance parameters estimated by benchmarking. For safety reasons, we have chosen a different approach using a *worst case execution time* (WCET) performance model: instruction cycles have been counted taking into account all the delays introduced by pipeline stalls, whenever they are possible to occur.

The ATU is a configurable unit featuring twelve channels. Each channel provides different specialized functions, including interval timer operations, free-running or cyclic counter operations, compare&match and input capture functions, PWM timer functions, event count and event cycle measurement, input edge measurement and edge in-

[2]This value refers to the functional description. Mapping a function on architectural resources might require to assign one or more CFSMs to two or more parts.

put cessation detection functions, noise cancellation. The ATU's capabilities are augmented by the possibility of combining one or more of the functions listed above to perform a more complex operation. For example, the ATU offers the capability of detecting crankshaft wheel teeth filtering out unwanted glitches, of counting teeth, of measuring tooth period, of signaling particular conditions (such as TDC occurrences), without intensive CPU involvement.

On the software side, the support needed for task execution, i.e. task scheduling and CPU assignment, is provided and modeled. It implies both IRQ management and the choice of a scheduling policy. At this level of abstraction, ISRs are modeled simply as *drivers* between the hardware and software layers: they detect the presence of an IRQ on the hardware side and pass the control to the RTOS. It in turns activates all the tasks sensitive to that IRQ, according to the selected scheduling policy. This scheme is also called *Deferred Service Routine*. The interrupt handlers are characterized by a priority level, set in the mapping phase, and an ISR overhead. Similarly, task scheduling is modeling the scheduling policy and the context switch overhead needed to start, suspend, resume and stop each task execution.

5. Architecture Exploration

The target architecture has been investigated using four different cost functions: *CPU load, interrupt frequency, task switching* and *task number* (strictly related to memory requirements). We have analyzed three different variations of the basic hw/sw architecture.

Mapping "A": a choice for software re-use. The first mapping is intended to allow software re-use and portability. In this choice, the software implementation has not been optimized for the Hitachi architecture. This means that we have not used some specific ATU features to have a more portable code. Thus we have employed only those ATU functions, such as input edge measurement and interval timer operations, likely to be provided also by other micro-controllers featuring a different, simpler timer unit.

In order to map the functional specification to the selected architecture, we decomposed it to an equivalent description "mappable" onto the hardware and software partitions (*functional restructuring*). In this case, the mapped behavior consists of 13 CFSMs, 3 down-counters, 1 free-running counter and 1 input capture.

The task scheduling policy is chosen taking into account that task execution is fired by two asynchronous events, i.e. the crankshaft and the camshaft edges. Since the tooth frequency is about 60 times higher than the one of the camshaft edges, we scheduled statically all the software CFSMs in order to minimize both the task number and the context switch overhead.[3] On each crankshaft and camshaft edge every task is fired and eventually skipped if not to be executed.

[3]A static scheduled task chain is equivalent to a single, complex task.

Performing performance simulation, we have collected the data summarized in Table 1 (refer to mapping "A1") and analyzed task execution frequencies. The analysis reveals how some tasks are often skipped, since there is no input event firing them. A logical conclusion is that the CPU load could be reduced by reducing the number of task skipped on each execution.

In mapping "A2", we have removed from the static scheduled chains the tasks with the lowest activation frequency and introduced a second static scheduled chain consisting of the tasks with the minimum activation frequency. The two task chains are dynamically scheduled by a static priority scheduler and their priorities are assigned according to the Rate Monotonic Algorithm, i.e. more frequent tasks are given a higher priority. As shown in Table 1, the overhead introduced by the higher task switching is not worth the reduction in the number of skipped tasks. The overall CPU load is higher than solution "A1" and using a single static scheduler seems a better choice for the scheduling policy. In this work we have selected "manually", though with the aid of the tool, the "best" number of tasks. However, new techniques exist that allow, in some cases, this choice to be made automatically [4].

Mapping "B": exploiting the ATU capabilities. The second solution tries to minimize the amount of software executed by the CPU by exploiting all the ATU's functions. The application has been completely customized on the Hitachi architecture, using the noise cancelling, the compare&match and the event counting features. The hardware itself counts fly-wheel teeth, detects, for example, TDC occurrences and generates the events on which to fire the software tasks. Because of the features of this architecture, we have to re-partition the behavior into a different set of CFSM to have the correct granularity that allows to fully exploit the power of the ATU. The "new" system behavior consists of 15 CFSMs, 3 down-counters, 1 input capture, 1 noise canceller, 1 compare&match and 1 TDC-period counter.

As stated before, TDCs are detected directly by the hardware and no other software intervention is needed to execute the periodic related tasks. Nevertheless, crankshaft signal filtering still requires one interrupt per tooth, and a single statically scheduled task chain (Mapping "B1") is again the best choice (compare the CPU load of mappings "B1" and "B2" in Table 1) as it was in the previous mapping.

Independently of the scheduling policy adopted, mapping "B" yields a reduction of CPU load, since several operations are mapped to the hardware partition. This is obtained mainly at the expense of interrupt frequency, needing a stronger communication between hardware and software. This could have been suggested by the finer decomposition of the functional specification.

Table 1. Mapping cost parameters

mapping	"C"	"B2"	"B1"	"A2"	"A1"
CPU load%	1.4	6.8	6.7	10.3	10.2
IRQs	1528	8626	8626	7739	7739
task switching	1673	7745	7616	7745	7616
task number	6	3	2	3	2

Mapping "C": adding hardware. Finally we have accounted for the possibility of customizing the selected architecture for the engine control application. This imply the addition of hardware to enhance the ATU's capabilities, as required by the specific application. As suggested by the previous analysis, we have to tackle the fly-wheel signal filtering issue, which is the key for cutting down CPU load and interrupt frequency. Hence we have added virtual (i.e. still not designed) hardware acting as crankshaft signal filter to the selected architecture and analyzing its performance effectiveness, to see if it is worth the cost and the work required. Once again, to evaluate this new solution, we have modified the behavioral description ending with 18 CFSMs, 2 down-counters, 1 input capture, 1 noise canceller, 1 compare&match, 1 TDC-period counter.

With regard to the task scheduling policy, in this case we can optimize task activation by exploiting the asynchronous events generated by the hardware (e.g. the TDC events), instead of using a single, statically scheduled task chain. A static priority[4] dynamic scheduler has been adopted to schedule 2 static scheduled task chains and 4 other tasks. This results in a higher task number, but yields a very large improvement in all other performance parameters, as shown in Table 1. We have compared the different mappings referring to global performance metrics. However, performance simulation allows us to analyze information such as the latency time of some critical operations: for example, RPM computation. The mean time between the activation and the RPM output varies from 61μs for mapping "A1" to 53μs for mapping "C". Figure 4 shows an instance of the activation/reaction points, and the reaction time, as measured for mapping "C". The RPM is computed by the static scheduled task chain "0", fired by the related ISR, according to the deferred interrupt scheme used. In this case, since no other interrupt occurs and no higher priority task has to be executed, the overall latency time is the sum of the interrupt completion overhead (estimated in 0.7μs), RTOS start overhead (about 10μs) and task chain execution time. A similar analysis can be carried out for each task and/or for each activation in order to verify that all the deadlines are met.

[4] Priorities have been assigned using the Rate Monotonic Algorithm.

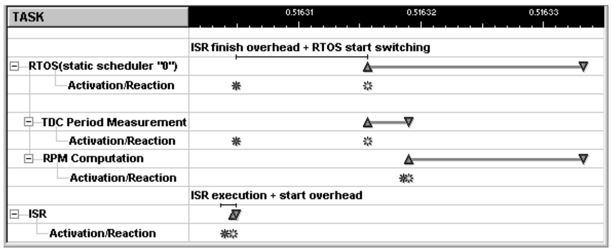

Figure 4. Gantt chart for the RPM computation

6. Conclusions

In this paper we have applied a *functional/architecture* co-design methodology to an engine management system. The use of the Felix VCC framework has allowed the exploration of different architectures and mapping, by a "tool-driven" search in the solution space, in a total of two week/man. In the presented case study, we have found the "optimal" mapping to the selected hardware architecture, based on the chosen cost functions, and we have identified those system functions candidates for hardware implementation, evaluating their performance impacts before implementation. The tool has also been used to estimate the impact of different CPUs on system performance and it is currently being used to evaluate the pros and cons of a novel dual processor architecture for power-train control functions.

Acknowledgments. Walter Nesci and Giorgio Bombarda of Magneti Marelli, Luciano Lavagno of Cadence Berkeley Lab, Frank Schirrmeister and all the Felix/VCC team have contributed valuable comments and suggestions.

References

[1] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS Approach*. Kluwer Academic Publishers, 1997.

[2] Hitachi. *SH7055. Hardware Manual*, March 1998.

[3] G. Martin and B. Salefski. Methodology and technology for design of communications and multimedia products via system-level ip integration. In *Proceedings of the Design Automation Conference*, 1998.

[4] Y. Watanabe, M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proceedings of the Design Automation Conference*, pages 805–810, 1999.